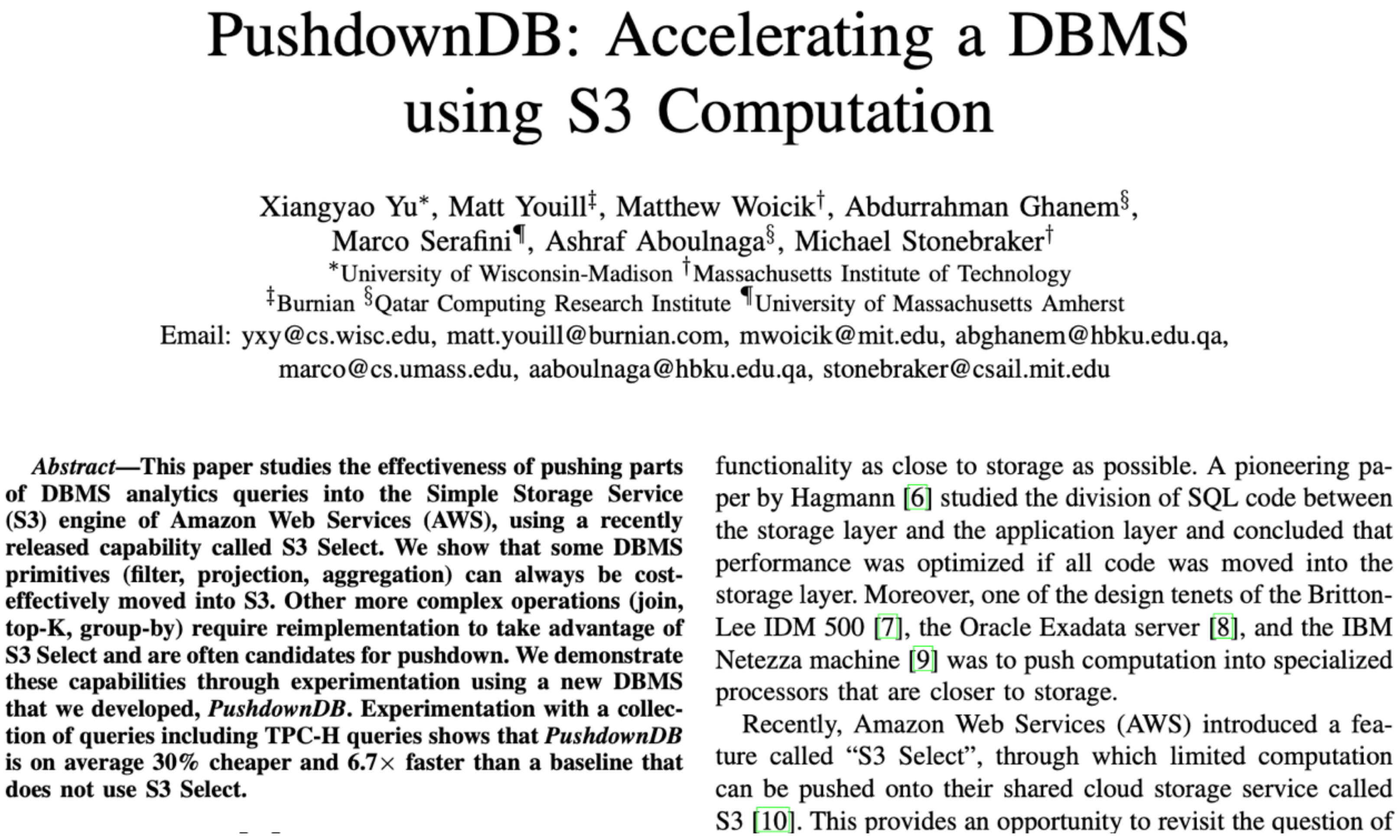
WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

CS 764: Topics in Database Management Systems

Lecture 21: PushdownDB

Xiangyao Yu
11/16/2020

Today’s Paper



PushdownDB: Accelerating a DBMS using S3 Computation

Xiangyao Yu*, Matt Youill‡, Matthew Woicik†, Abdurrahman Ghanem§,
Marco Serafini¶, Ashraf Aboulnaga§, Michael Stonebraker†
*University of Wisconsin-Madison †Massachusetts Institute of Technology
‡Burnian §Qatar Computing Research Institute ¶University of Massachusetts Amherst
Email: yxy@cs.wisc.edu, matt.youill@burnian.com, mwoicik@mit.edu, abghanem@hbku.edu.qa,
marco@cs.umass.edu, aaboulnaga@hbku.edu.qa, stonebraker@csail.mit.edu

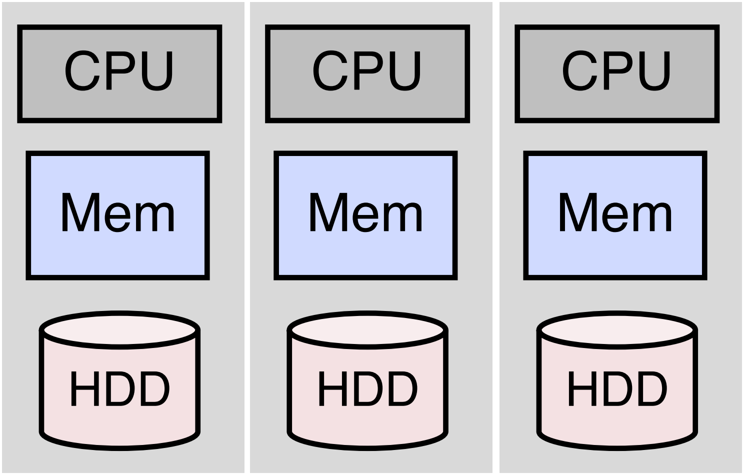
***Abstract*—This paper studies the effectiveness of pushing parts of DBMS analytics queries into the Simple Storage Service (S3) engine of Amazon Web Services (AWS), using a recently released capability called S3 Select. We show that some DBMS primitives (filter, projection, aggregation) can always be cost-effectively moved into S3. Other more complex operations (join, top-K, group-by) require reimplementation to take advantage of S3 Select and are often candidates for pushdown. We demonstrate these capabilities through experimentation using a new DBMS that we developed, *PushdownDB*. Experimentation with a collection of queries including TPC-H queries shows that *PushdownDB* is on average 30% cheaper and 6.7× faster than a baseline that does not use S3 Select.**

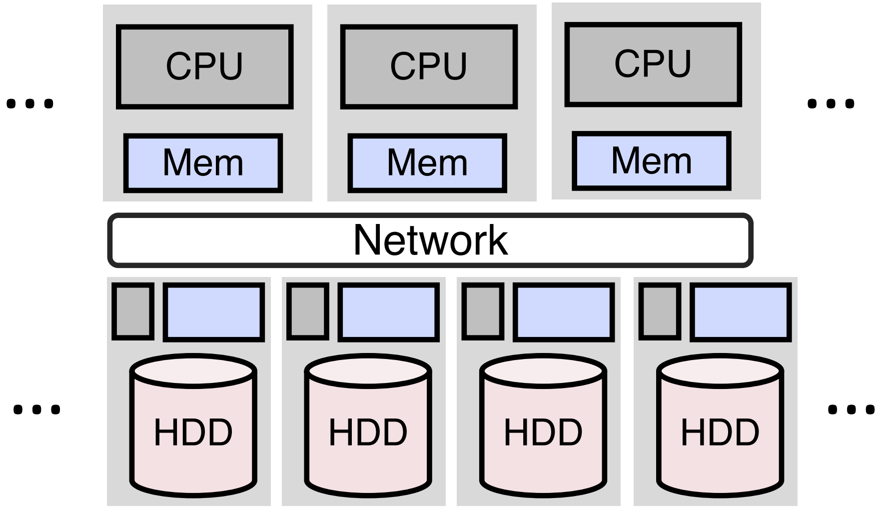
functionality as close to storage as possible. A pioneering paper by Hagmann [6] studied the division of SQL code between the storage layer and the application layer and concluded that performance was optimized if all code was moved into the storage layer. Moreover, one of the design tenets of the Britton-Lee IDM 500 [7], the Oracle Exadata server [8], and the IBM Netezza machine [9] was to push computation into specialized processors that are closer to storage.

Recently, Amazon Web Services (AWS) introduced a feature called “S3 Select”, through which limited computation can be pushed onto their shared cloud storage service called S3 [10]. This provides an opportunity to revisit the question of

ICDE 2020

Cloud Database Architecture

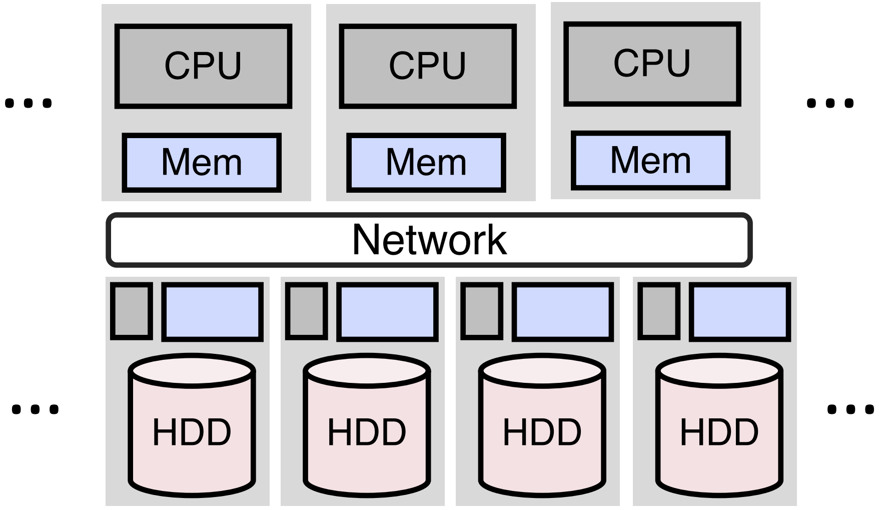




On-premises
- Fixed and limited hardware resources
- Shared-nothing architecture

Cloud
- Virtually infinite computation & storage, Pay-as-you-go price model
- Disaggregation architecture

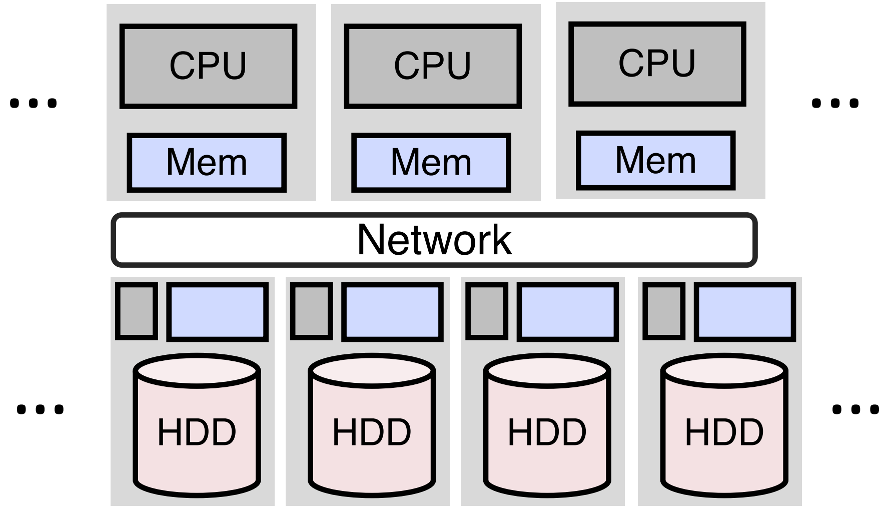
Storage-Disaggregation Architecture



Features of the disaggregation architecture

- Computation and storage layers are disaggregated
- Limited computation can happen in the storage layer

Storage-Disaggregation Architecture



Features of the disaggregation architecture

- Computation and storage layers are disaggregated
- Limited computation can happen in the storage layer

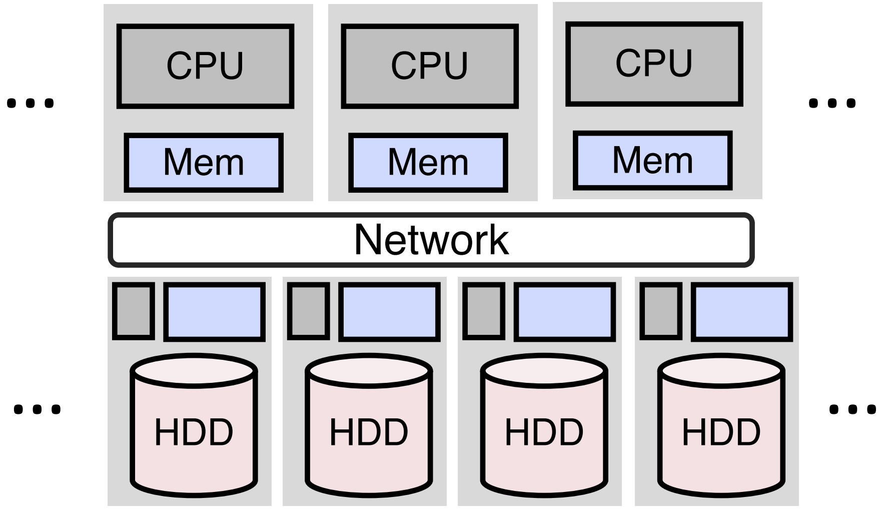
Advantages

- Lower management cost
- Independent scaling of computation and storage

Disadvantages

- **Network becomes a bottleneck**

How to Mitigate the Network Bottleneck?



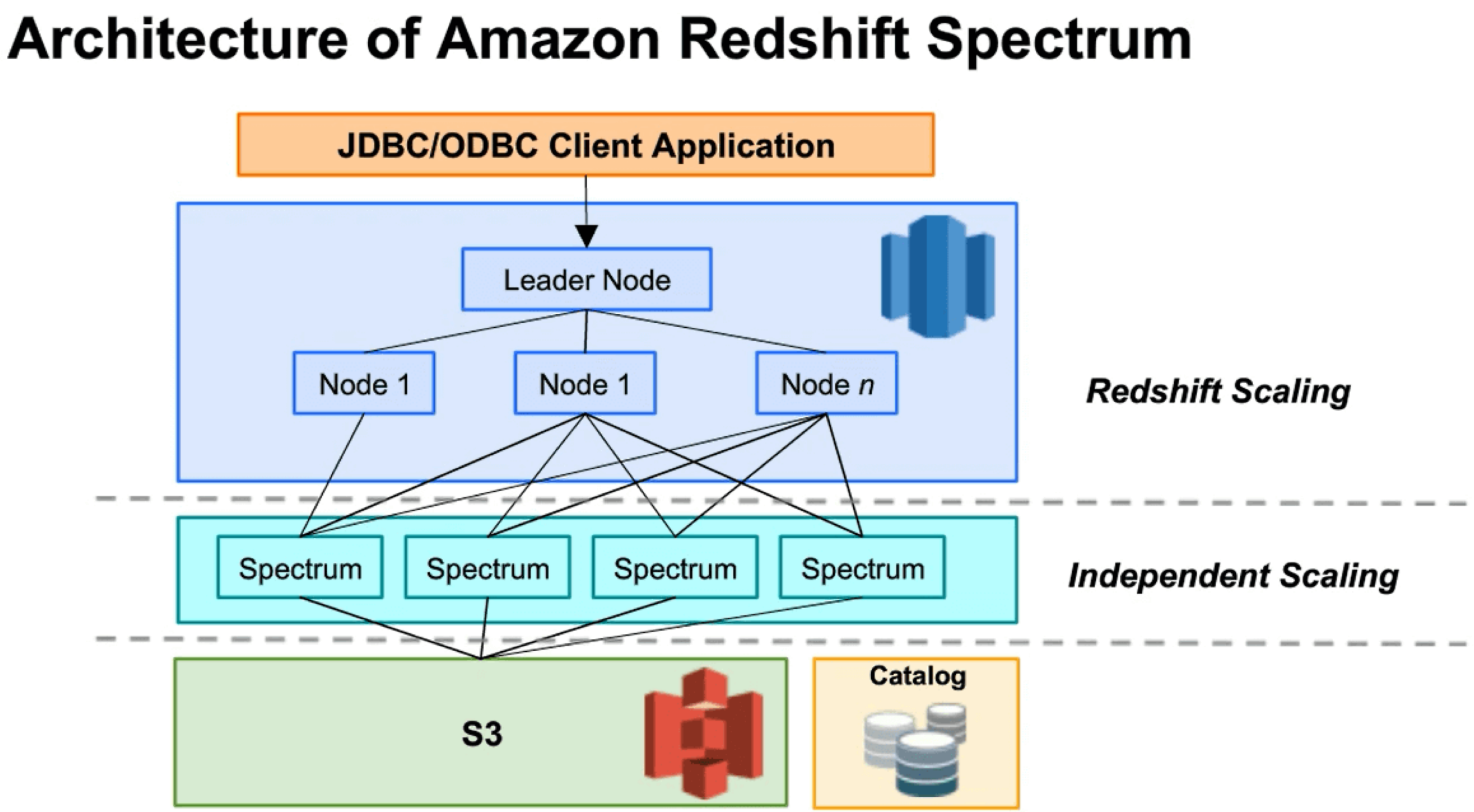
Solution 1: Move data to computation

- Cache storage data in the computation layer
- Example: Snowflake

Solution 2: **Move computation to data**

- Pushdown computation to the storage layer
- Example: PushdownDB (this talk)

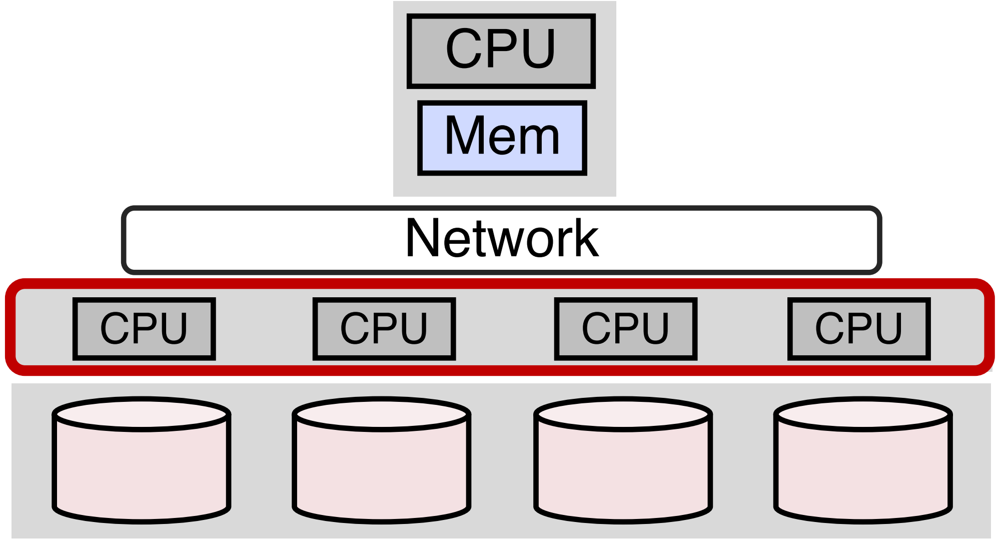
What about Redshift Spectrum?



The Redshift layer is similar to static caching

The Spectrum layer implements computation pushdown

PushdownDB Architecture

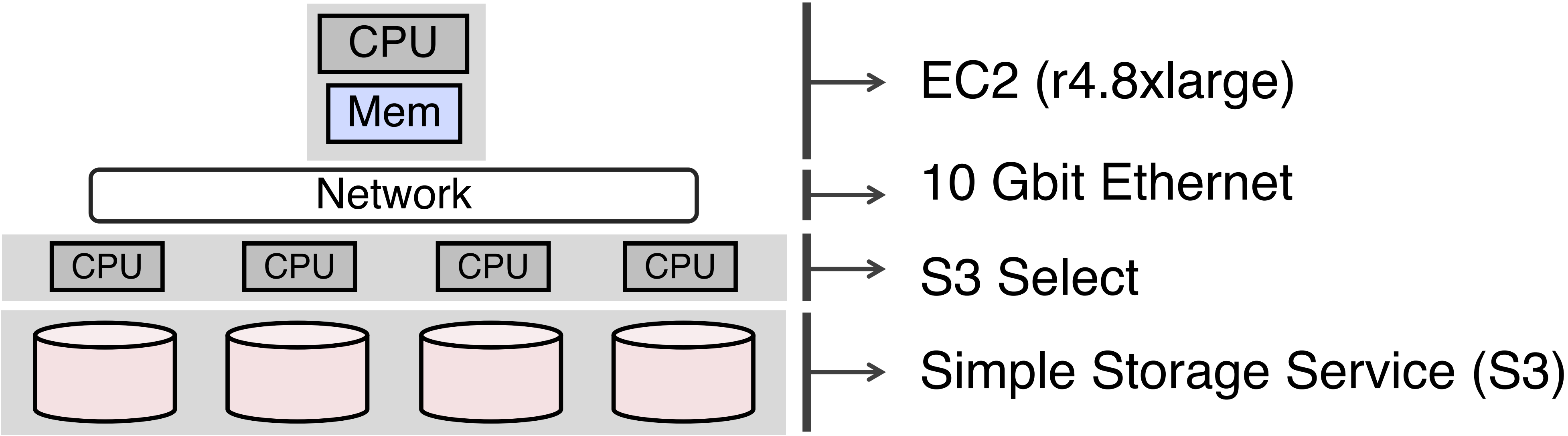


Main tenet of database systems: **Keep computation close to storage**

Key questions to address in this project:

- How to implement relational operators to leverage existing cloud services?
- What are the performance and cost tradeoffs?

PushdownDB – Building Blocks



EC2 (r4.8xlarge)

10 Gbit Ethernet

S3 Select

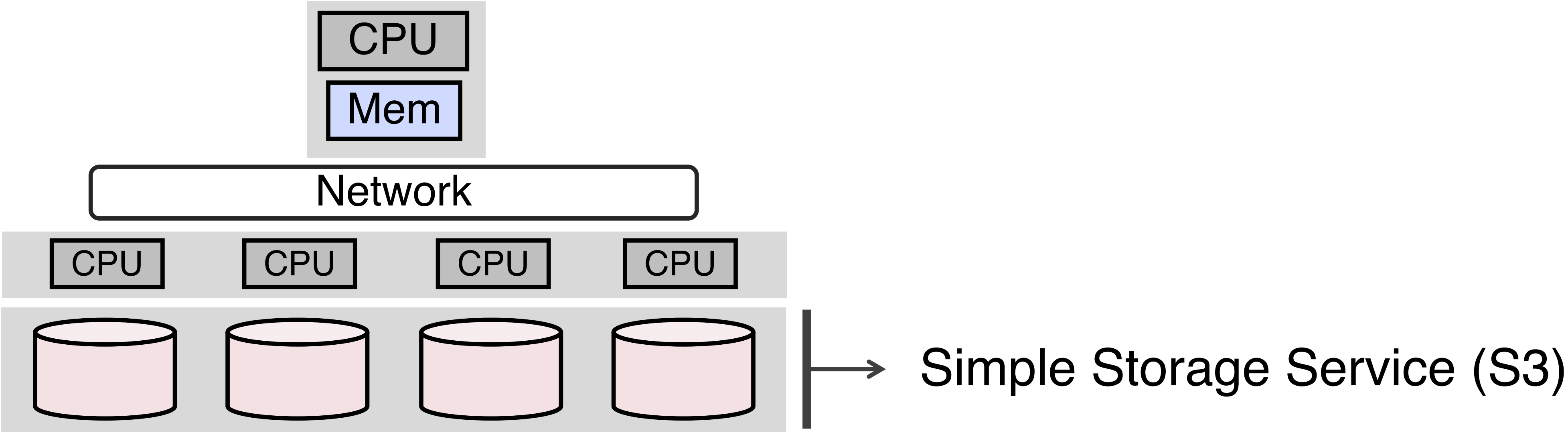
Simple Storage Service (S3)

PushdownDB implementation

- Single-node, multi-process Python-based database
- Ubuntu 16.04.5 LTS, Python version 2.7.12.

Source code: https://github.com/yxymit/s3filter.git

Simple Cloud Storage (S3)



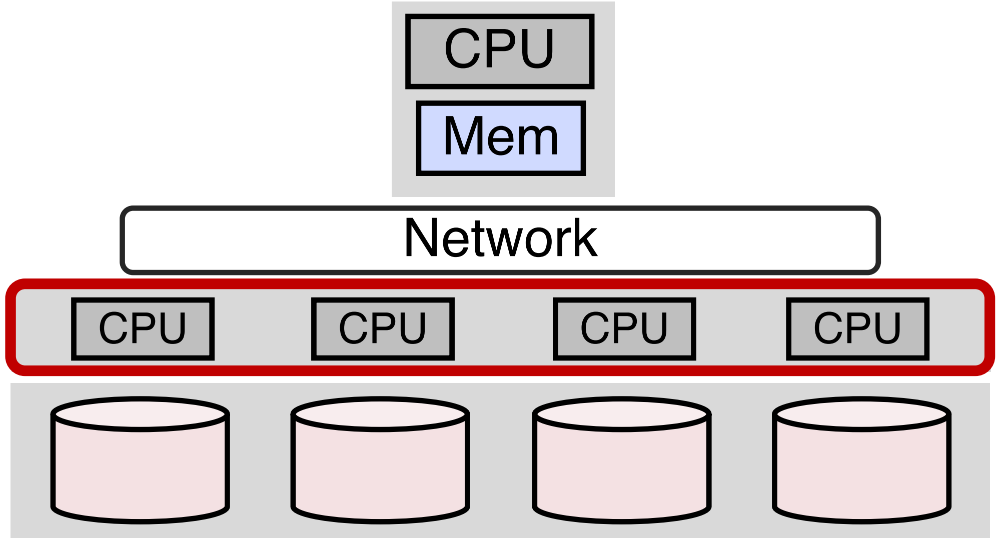
Virtually infinite storage capacity with relatively low cost

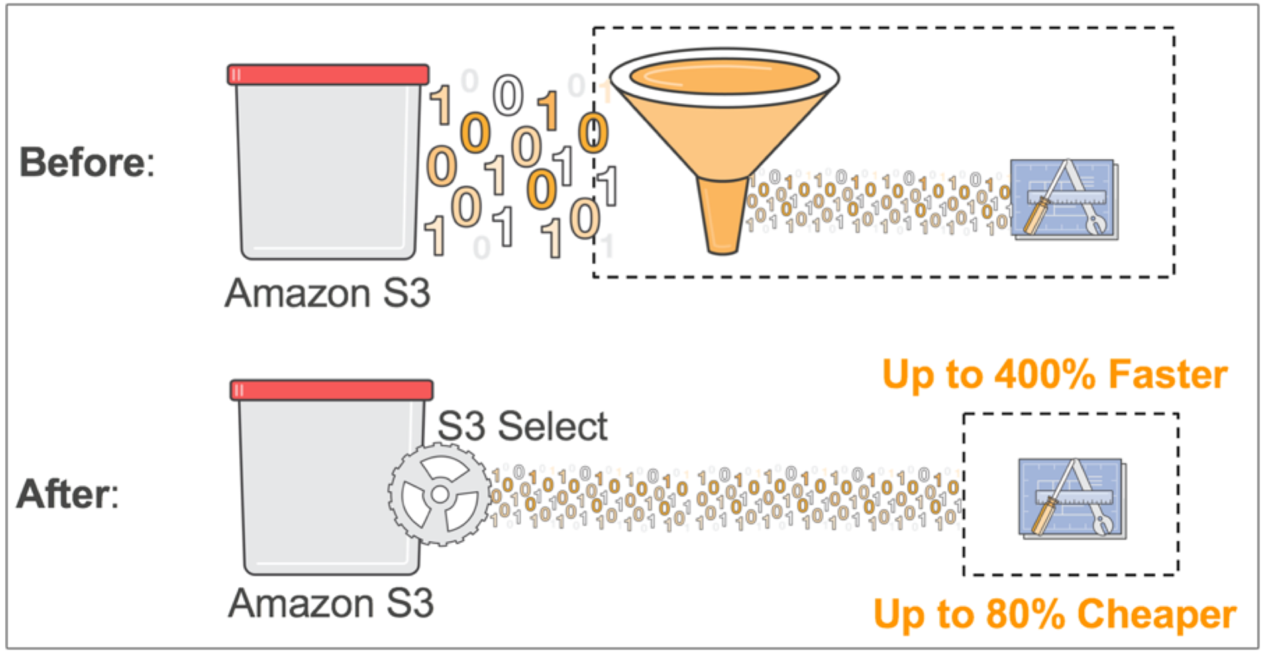
Partition input relations into multiple shards, each shard is stored as a separate object in S3

S3 vs. elastic block store (EBS) vs. local store

- Virtually infinite capacity, shared across all nodes, lower cost, durable

S3 Select





Supports limited SQL queries on **CSV** and **Parquet** data format

- S3 Select recognizes database schema for both data formats
- Simple queries with predicates and aggregation (no join, no group-by, no sort, etc.)

Cost Factors

Storage cost

- $0.022/GB/month for data in S3

Data transfer cost

- Free for data transfer within the same region
- $0.09/GB for transferring data out of AWS

S3 select cost

- Data scan cost: $0.002 per GB
- Data return cost: $0.0007 per GB

Network request cost

- $0.0004 per 1,000 requests

Computation cost

- Depends on the instance type
- For r4.8xlarge (32 core, 244 GB of memory), $2.128 per instance per hour

Cost Factors

Storage cost

- $0.022/GB/month for data in S3

Data transfer cost

- Free for data transfer within the same region
- $0.09/GB for transferring data out of AWS

S3 select cost

- **Data scan cost**: $0.002 per GB
- **Data return cost**: $0.0007 per GB

Network request cost

- $0.0004 per 1,000 requests

Computation cost

- Depends on the instance type
- For r4.8xlarge (32 core, 244 GB of memory), $2.128 per instance per hour

PushdownDB – Supported Operators

S3 Select supports

- Filter
- Project
- Aggregate without group-by

PushdownDB supports

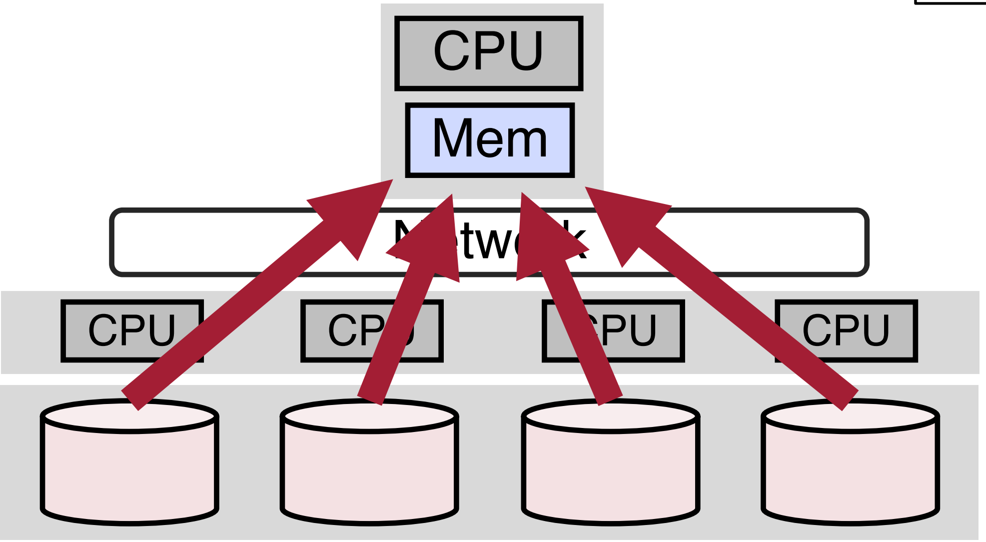
- Filter
- Project
- **Top-K**
- **Join**
- **Group-by**

Filter

Server-side filtering

- Compute server loads entire table from S3 and filters locally

Example query:
SELECT col1, col2
FROM R
WHERE col1 < 10



Filter

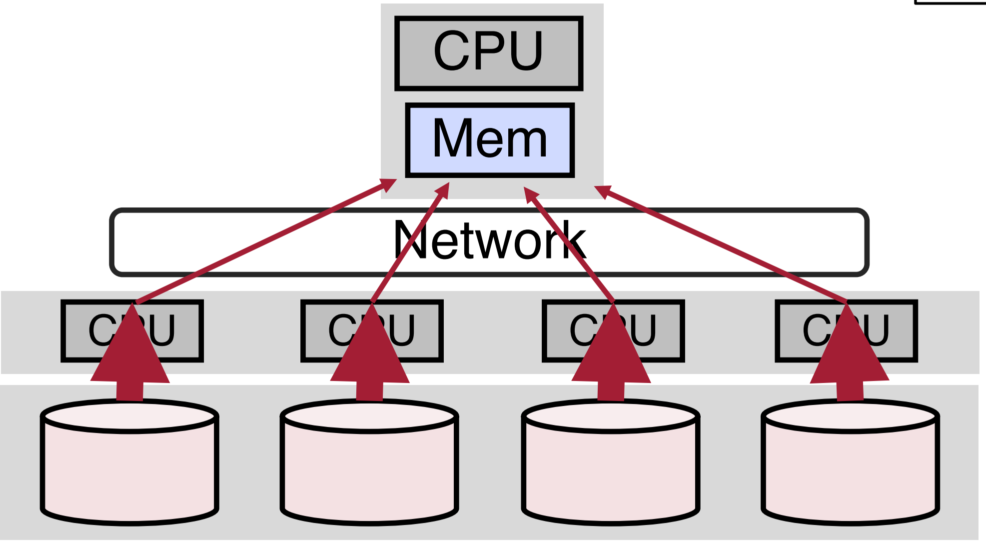
Server-side filtering

- Compute server loads entire table from S3 and filters locally

S3-side filtering

- Push down predicate evaluation using S3 Select

Example query:
SELECT col1, col2
FROM R
WHERE col1 < 10



Filter

Server-side filtering

- Compute server loads entire table from S3 and filters locally

S3-side filtering

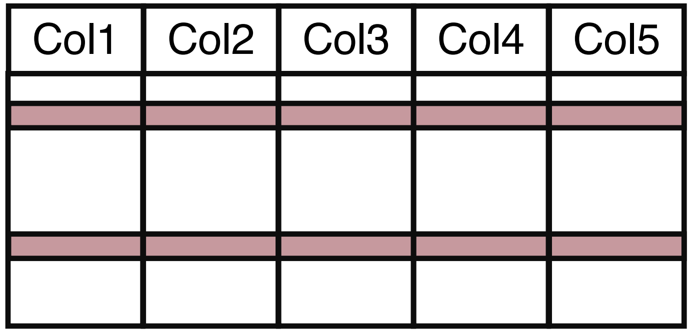
- Push down predicate evaluation using S3 Select

Example query:

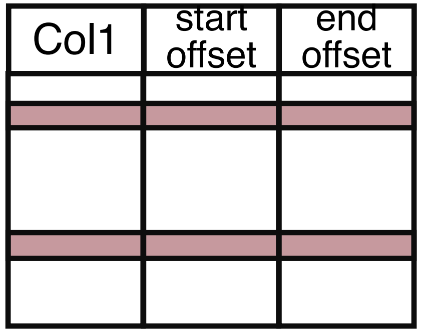
```
SELECT col1, col2
FROM R
WHERE col1 < 10
```

Indexing

- Push down predicate evaluation using S3 Select

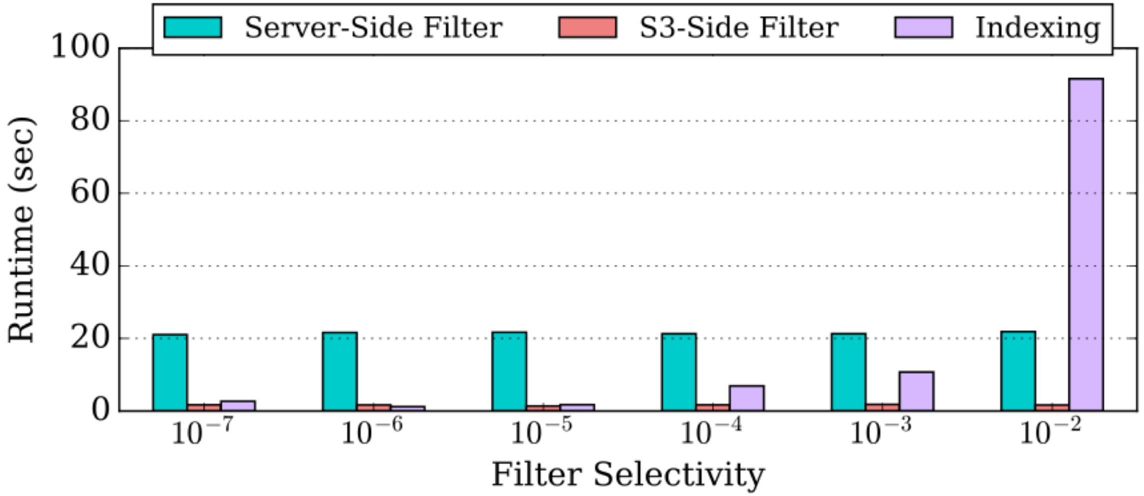


Original Table

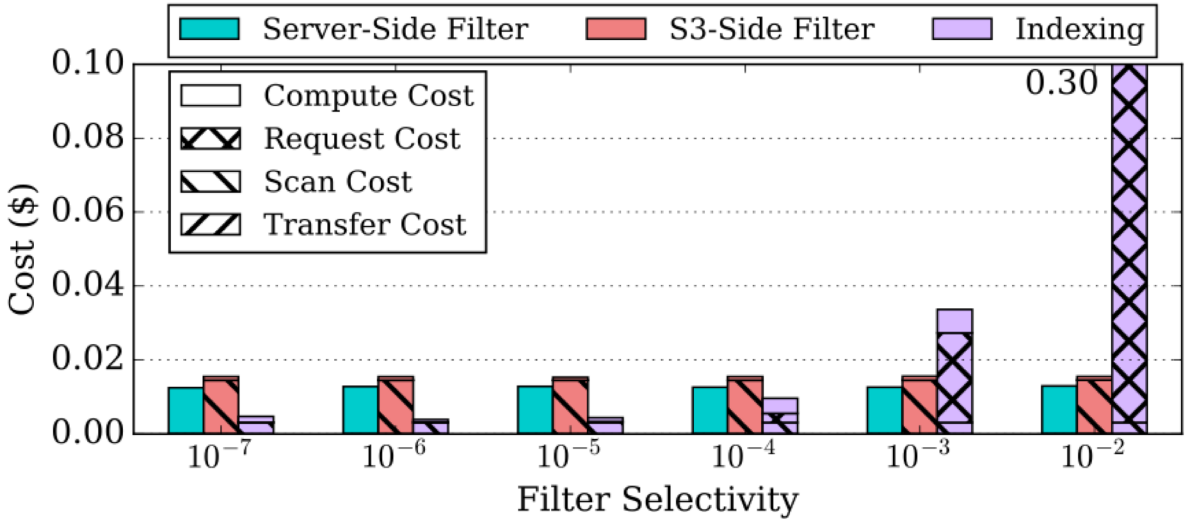


Index Table

Filter – Evaluation



(a) Runtime



(b) Cost

Join

Baseline Join

- Server loads both tables from S3 and joins locally

```
SELECT SUM(O_TOTALPRICE)
FROM CUSTOMER, ORDER
WHERE
   O_CUSTKEY = C_CUSTKEY
   AND C_ACCTBAL <= upper_c_acctbal
   AND O_ORDERDATE < upper_o_orderdate
```

Join

Baseline Join

- Server loads both tables from S3 and joins locally

Filtered Join

- Server pushes filtering predicates to S3 to load both tables

```
SELECT SUM(O_TOTALPRICE)
FROM CUSTOMER, ORDER
WHERE
   O_CUSTKEY = C_CUSTKEY
   AND C_ACCTBAL <= upper_c_acctbal
   AND O_ORDERDATE < upper_o_orderdate
```

Join

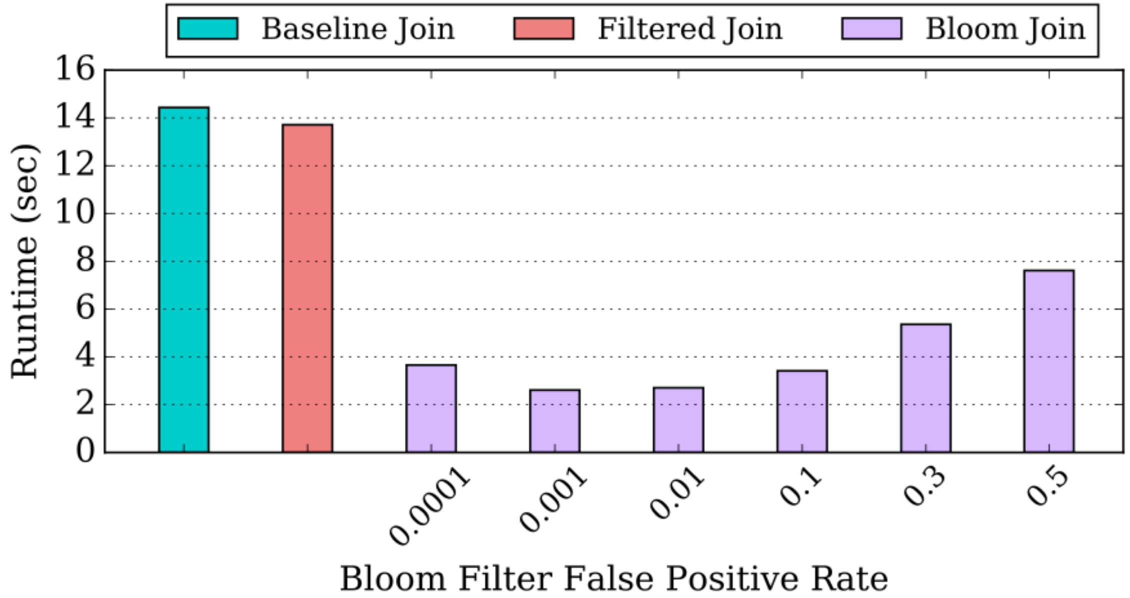
Bloom Join

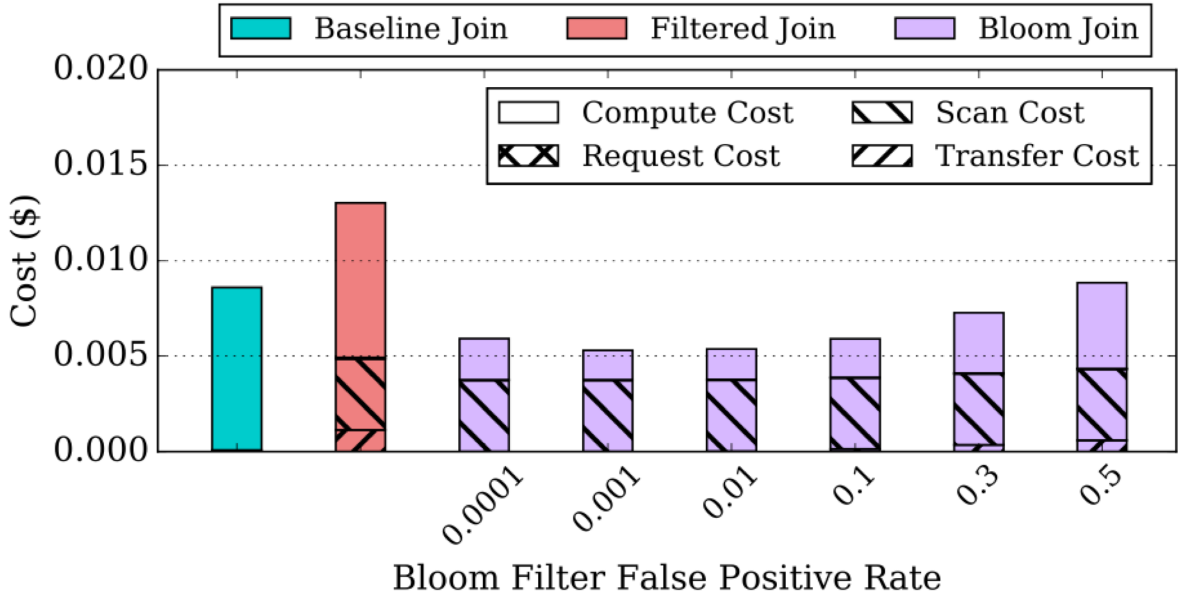
- Step 1: Server loads the smaller table, builds a bloom filter using join key
- Step 2: Server sends the filter via S3 Select to load the bigger table
- Bloom filter is pushed down as a predicate

```
SELECT ...
FROM S3Object
WHERE SUBSTRING('1000011...111101101',
      ((69 * CAST(attr as INT) + 92) % 97) % 68 + 1, 1 ) = '1'
```

```
SELECT SUM(O_TOTALPRICE)
FROM CUSTOMER, ORDER
WHERE
   O_CUSTKEY = C_CUSTKEY
   AND C_ACCTBAL <= upper_c_acctbal
   AND O_ORDERDATE < upper_o_orderdate
```

Join – Evaluation





```
SELECT SUM(O_TOTALPRICE)
FROM CUSTOMER, ORDER
WHERE
   O_CUSTKEY = C_CUSTKEY
   AND C_ACCTBAL <= upper_c_acctbal
   AND O_ORDERDATE < upper_o_orderdate
```

Group-By

Example
group-by query

```
SELECT c_nationkey, sum(c_acctbal)
FROM customer
GROUP BY c_nationkey;
```

Group-By

Example
group-by query

```
SELECT c_nationkey, sum(c_acctbal)
FROM customer
GROUP BY c_nationkey;
```

Server-Side Group-By

- Compute server loads entire table from S3 and performs group-by

Filtered Group-By

- Pushes filtering predicates to S3 when loading the table

Group-By

Example
group-by query

```
SELECT c_nationkey, sum(c_acctbal)
FROM customer
GROUP BY c_nationkey;
```

S3-Side Group-By

- Step 1: S3 Select to obtain all c_nationkey values
- Step 2: Load data from S3 through the following S3 Select

```
SELECT
   sum(CASE WHEN c_nationkey = 0 THEN c_acctbal ELSE 0 END),
   sum(CASE WHEN c_nationkey = 1 THEN c_acctbal ELSE 0 END)
   ...
FROM customer;
```

Group-By

Example group-by query

```
SELECT c_nationkey, sum(c_acctbal)
FROM customer
GROUP BY c_nationkey;
```

S3-Side Group-By

- Step 1: S3 Select to obtain all c_nationkey values
- Step 2: Load data from S3 through the following S3 Select

```
SELECT
   sum(CASE WHEN c_nationkey = 0 THEN c_acctbal ELSE 0 END),
   sum(CASE WHEN c_nationkey = 1 THEN c_acctbal ELSE 0 END)
   ...
FROM customer;
```

- **Limitation**: Significant computation in S3 if many groups exist

Group-By

Example group-by query

```
SELECT c_nationkey, sum(c_acctbal)
FROM customer
GROUP BY c_nationkey;
```

S3-Side Group-By

Hybrid Group-By

- Step 1: S3 Select to obtain all c_nationkey values (can skip if histograms are available)
- Step 2: Perform S3-side group by for **only populous groups**

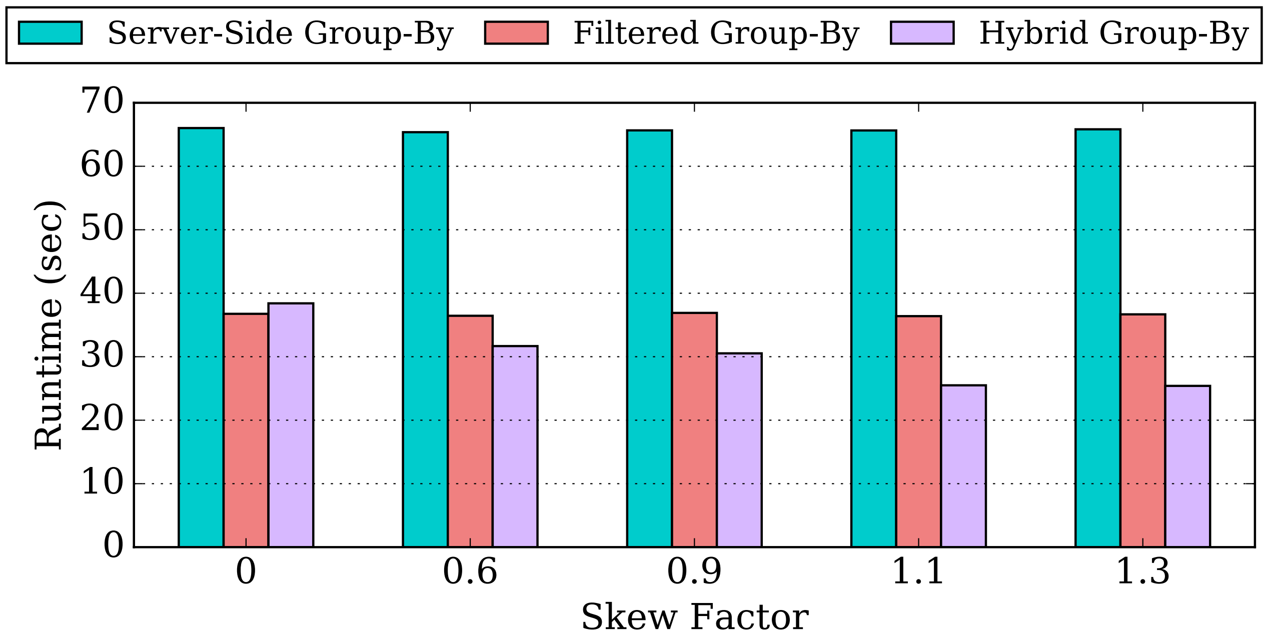
```
SELECT
   sum(CASE WHEN c_nationkey = 0 THEN c_acctbal ELSE 0 END)
FROM customer;
```

- Step 3: In parallel to Step 2, load the rest columns to server and performs group-by locally

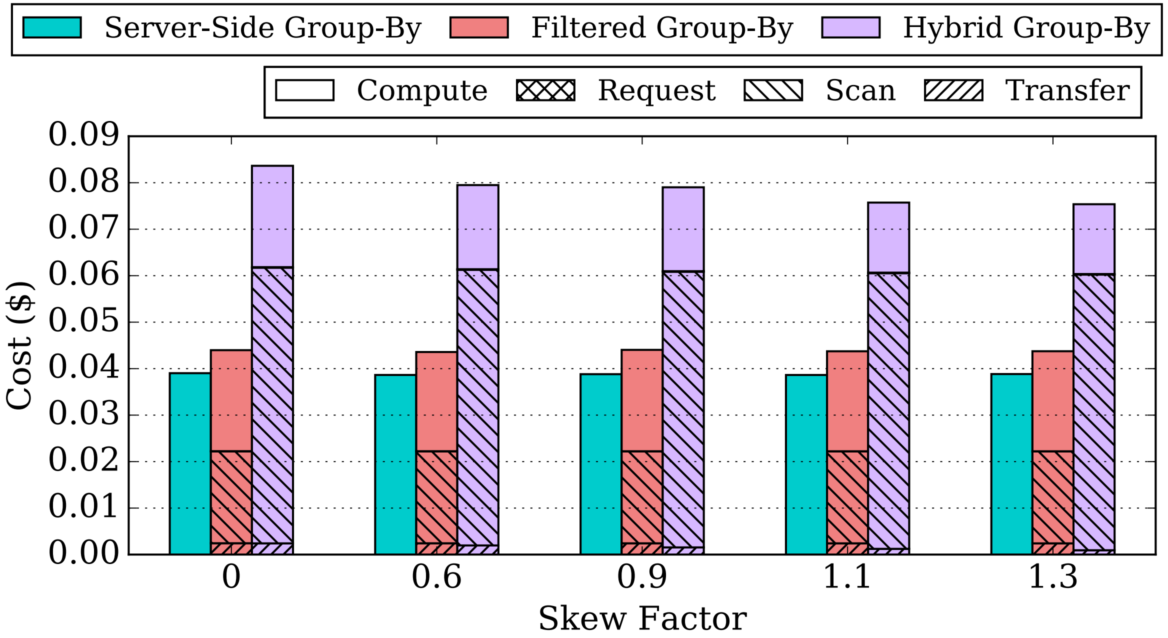
```
SELECT c_nationkey, sum(c_acctbal)
FROM customer
WHERE c_nationkey <> 0;
```

Group-By – Evaluation

Runtime



Cost Breakdown



Hybrid group-by **reduces runtime by 31%**

Hybrid group-by **increases cost due to multiple scans**

Top-K

Example
top-K query

```
SELECT *
FROM lineitem
ORDER BY l_extendedprice ASC
LIMIT K
```

Server-Side Top-K

- Compute server loads entire table from S3 and performs top-K

Top-K

Example
top-K query

```
SELECT *
FROM lineitem
ORDER BY l_extendedprice ASC
LIMIT K
```

Sampling-based top-K

- Observation: if have seen K values less than a threshold, there is no need to load values greater than the threshold

Top-K

Example
top-K query

```
SELECT *
FROM lineitem
ORDER BY l_extendedprice ASC
LIMIT K
```

Sampling-based top-K

- Phase 1: Load a sample of S records (load only the columns in ORDER BY clause) and calculate the threshold
- Phase 2: Load all records that are smaller than the threshold

Top-K

Example top-K query

```
SELECT *
FROM lineitem
ORDER BY l_extendedprice ASC
LIMIT K
```

Sampling-based top-K

- Phase 1: Load a sample of S records (load only the columns in ORDER BY clause) and calculate the threshold
- Phase 2: Load all records that are smaller than the threshold

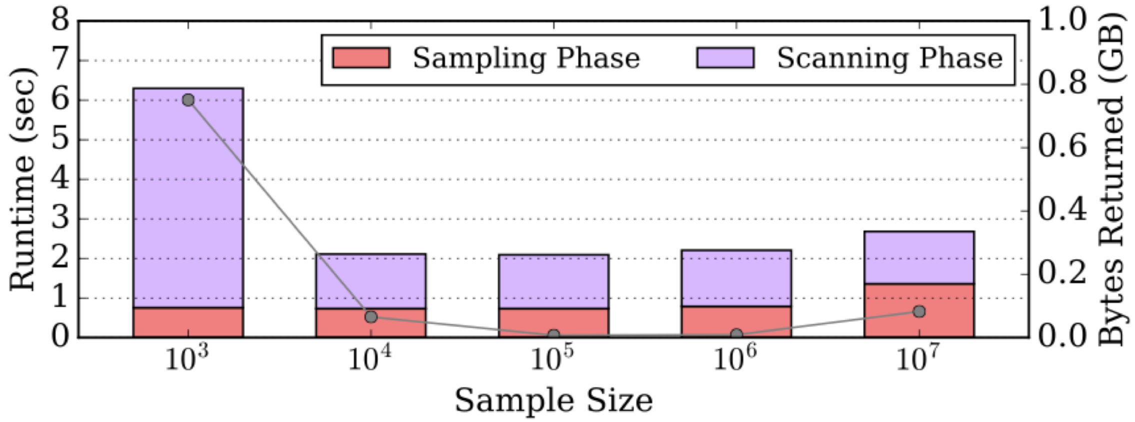
Total network traffic: $D = D_1 + D_2 = \alpha SB + \frac{KNB}{S}$

- B: size of each row in bytes
- S: the sample contains S rows
- α: fraction of a row for the sampling phase
- N: table contains N rows

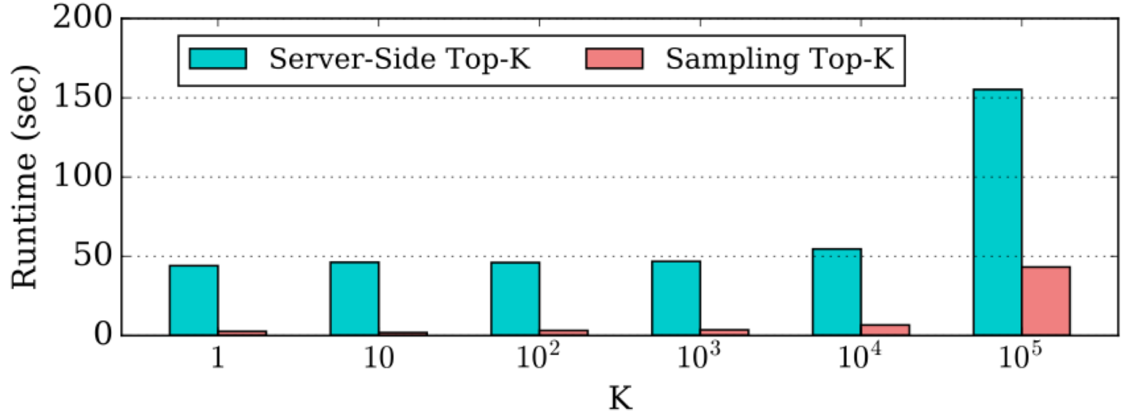
D is minimized when

$$S = \sqrt{\frac{KN}{\alpha}}$$

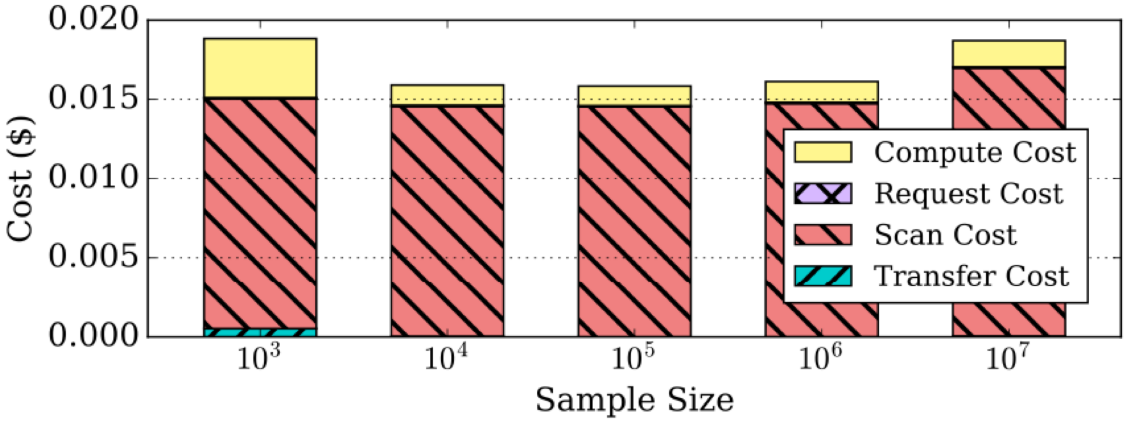
Top-K Evaluation



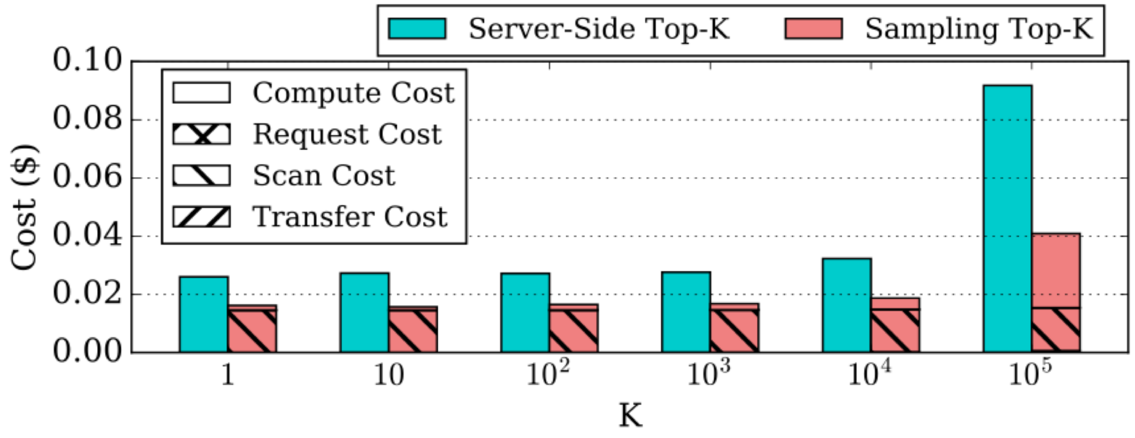
(a) Runtime



(a) Runtime

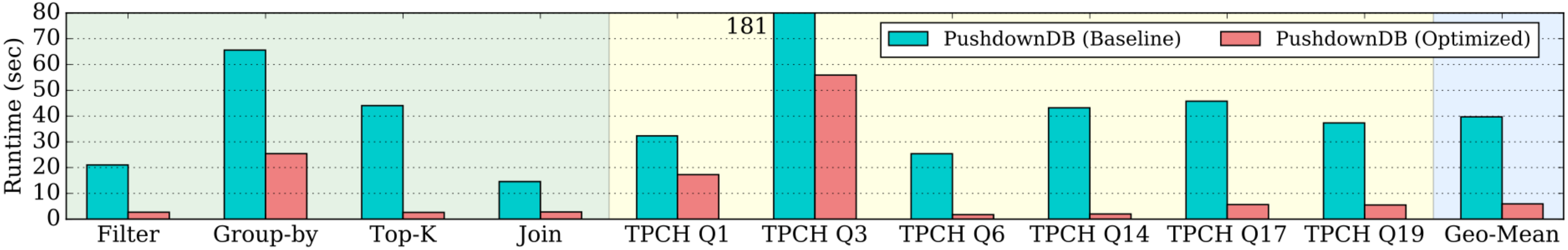


(b) Cost

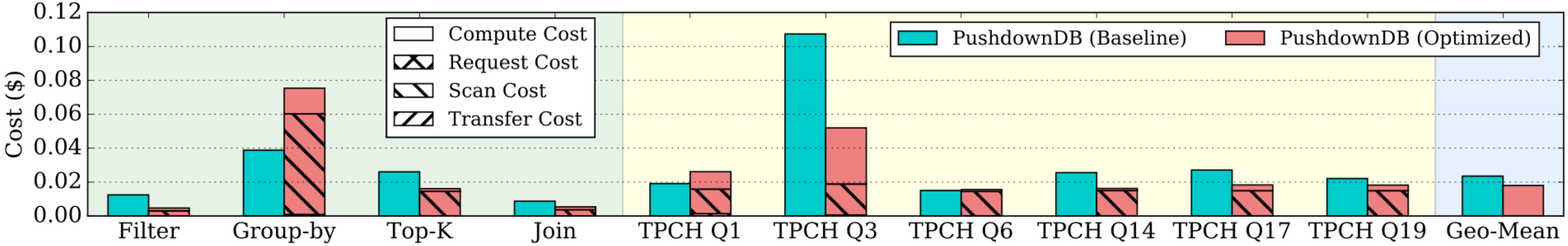


(b) Cost

Evaluation – All Operators and TPC-H



(a) Runtime



(b) Cost

Overall, PushdownDB **reduces runtime by 6.7×** and **reduces cost by 30%**

Discussion and Ongoing Work

Suggestions for S3 Select

1. Multiple byte ranges for each GET request
2. Index inside S3
3. More efficient Bloom filters
4. Partial group-by
5. Computation-aware pricing

Ongoing development on PushdownDB

- Rewrite the framework with C++
- Hybrid caching and pushdown
- Workload-specific caching policy

PushdownDB – Q/A

For the bloom filter, why need *k* hash functions instead of *1*?

How to handle node failures?

What if the compute node runs out of memory?

Competitor to S3 select outside of Amazon?

Would the indexing technique work for Snowflake as well?

Can operator pushdown be extended to other systems?

How stable are AWS prices?

Can a shared-nothing architecture perform better?

PushdownDB Discussion

Is it a good idea to entirely push the join operator to the storage layer? What are the main benefits and limitations of doing this?

Can you think of other aspects of databases (i.e., besides operator pushdown) or other applications that can also benefit from the storage-disaggregation architecture?

Next Lecture

Submit discussion summary to https://wisc-cs764-f20.hotcrp.com

- Title: **Lecture 21 discussion. group ##**
- Authors: Names of students who joined the discussion

Deadline: Tuesday 11:59pm

Submit review for

- Verbitski, Alexandre, et al., Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases, SIGMOD 2017